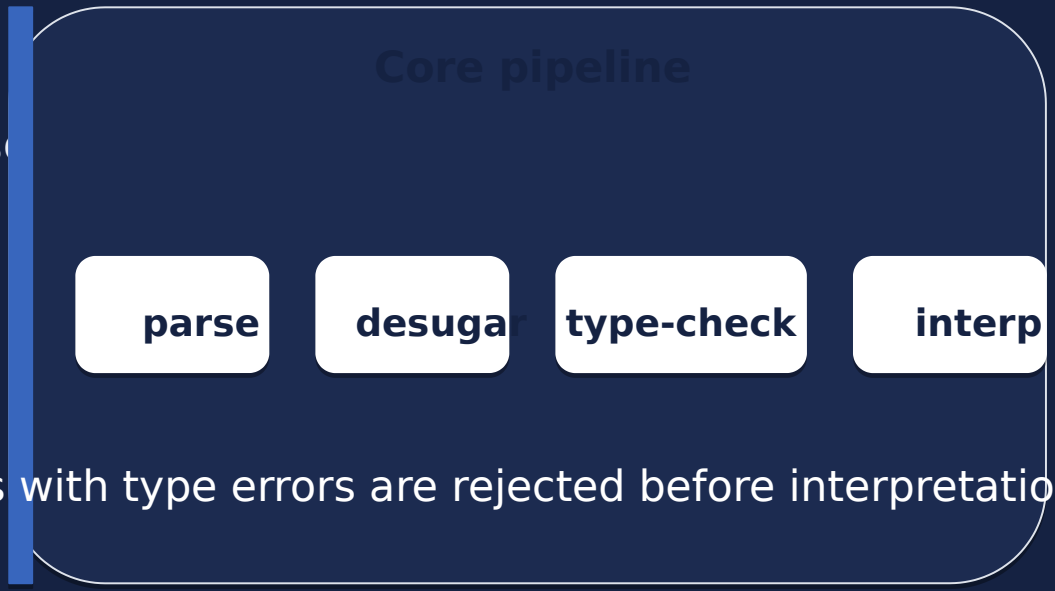
Mini Project 3

Type Checker Layer for SMoL-X

ntation overview

ld static checking without changing runtime evaluation s
: parse → desugar → type-check → interp
choice: if conditions must have type Bool



Programs with type errors are rejected before interpretation begin

Required changes

What had to change in the implementation

Type language

Add Type = Num | Bool | (Type -> Type)

Type environment

Add TypeEnv mapping Symbol → Type

Type-check pass

Add recursive type-check over the desugared core AST

Runner update

Change pipeline to parse → desugar → type-check → interp

Architecture and flow

The checker mirrors the interpreter but computes types instead of values

Pipeline view

parse: surface syntax → AST
desugar: surface sugar → core AST
type-check: core AST → Type
interp: core AST → Value

The checker runs after desugaring, so it only needs rules for the core language.

Parallel structure

Interpreter: Exp × Env → Value
Type checker: Exp × TypeEnv → Type
Both recurse over the same AST
One computes runtime results
The other computes static descriptions

Key implementation snippets

Core additions for MP3

Type definition

```
(define-type Type
  [numT]
  [boolT]
  [funT (argT : Type) (retT : Type)])
```

Type environment

```
(define-type-alias TypeEnv (Hashof Symbol Type))
(define mt-tenv (hash empty))
(define (extend-type tenv x t)
  (hash-set tenv x t))
```

Updated runner

```
(define (run [s : S-Exp]) : Value
  (let ([core (desugar (parse s))])
    (begin
      (type-check core mt-tenv)
      (interp core mt-env))))
```

Core shape of type-check

```
(define (type-check [e : Exp] [tenv : TypeEnv]) : Type
  (type-case Exp e
    [(numE n) (numT)]
    [(boolE b) (boolT)]
    [(varE x) (lookup-type x tenv)]
    [(plusE l r) ...]
    [(ifE test thn els) ...]
    [(let1E x rhs body) ...]
    [(lamE x xTy body) ...]
    [(appE fun arg) ...]))
```

Typing rules

Each rule corresponds to a branch in the type checker

Expression	Rule	Result
number	Always well-typed	Num
true / false	Always well-typed	Bool
{+ e1 e2}	Both operands must be Num	Num
{if c t e}	c must be Bool; t and e must match	Shared branch type
variable	Must exist in TypeEnv	Stored type
{lam {x : T} body}	Check body in env extended with x : T	(T -> bodyType)
{f a}	f must be a function and a must match parameter type	Return type
{let1 {x e} body}	Bind x to the type of e, then check body	Body type

Testing results

Accepted programs continue normally; rejected programs stop before execution

Accepted

Well-typed arithmetic
- Type: Num • Value: 12

Function definition and application
- Type: Num • Value: 10

Higher-order function
- Type: Num • Value: 11

Boolean literal check
- Type: Bool

Rejected

Wrong operand type in +
- expected Num, got (Num -> Num)

Wrong function argument type
- function argument mismatch

Conditional branch mismatch
- Num vs (Num -> Num)

Unbound variable
- x not in TypeEnv

Non-boolean if test
- expected Bool, got Num

Scenario explanations

Why the tests succeed or fail

Arithmetic succeeds

```
{+ 3 {+ 4 5}}
```

Each + sees Num on both sides, so the whole expression is Num.

Function application succeeds

```
{double 5}
```

double has type (Num -> Num), and 5 matches the declared parameter type.

Addition with a lambda fails

```
{+ 1 {lam {x : Num} x}}
```

+ requires two numbers, but the right operand is a function.

Mismatched if branches fail

```
{if true 42 {lam {x : Num} x}}
```

The checker validates both branches, not just the runtime path.

Non-boolean if test fails

```
{if 1 2 3}
```

This implementation uses real booleans, not numeric truthiness.

Reflection and takeaways

What the type-checker layer adds

Main insight

e checker is a second interpreter.

e same AST.
arallel environment.
s types instead of values.
vative: it may reject a program that would happen to run

Soundness

Well-typed programs do not get stuck on the covered error classes.

Conservatism

The checker may reject a program that would run on one particular path.

Annotations

Lambda parameter annotations make static guarantees possible.